Inheritance of Dynamic Behavior
Development of a Groupware Editor

Twan Basten[1]* and Wil M.P. van der Aalst[2]

[1] Dept. of Electrical Engineering, Eindhoven University of Technology, The Netherlands
tbasten@ics.ele.tue.nl
[2] Dept. of Computing Science, Eindhoven University of Technology, The Netherlands
wsinwa@win.tue.nl

Abstract. One of the key issues of object-oriented modeling is *inheritance*. It allows for the definition of subclasses that inherit features of some superclass. Inheritance is well defined for static properties of classes such as attributes and methods. However, there is no general agreement on the meaning of inheritance when considering dynamic behavior of objects. This paper studies inheritance of dynamic behavior in a framework based on Petri nets. The notions of an object life cycle and inheritance between life cycles are defined. The inheritance relation is based on two fundamental concepts, namely *blocking* and *hiding* method calls. Several transformation rules are given to construct subclasses from a given superclass, thus allowing reuse of life-cycle specifications during a design. To show the validity of the approach, the results are applied to the development of a groupware editor.

Key words: object orientation – inheritance – object life cycle – dynamic behavior – Petri nets – computer supported cooperative work (CSCW)

1 Introduction

In software-engineering practice, the popularity of object-oriented modeling and design is increasing rapidly. Three methods are in common use: OMT [12], OOD [4], and UML [6]. One of the key issues in any object-oriented method is *inheritance*. The inheritance mechanism allows the user to specify a subclass that inherits features of some other class, its superclass. A subclass has the same features as the superclass, but in addition it may have some other features.

The concept of inheritance is well defined for *static* features of an object class, i.e., its methods and its attributes. However, a class also contains a definition of the dynamic behavior of objects. That is, it specifies the order in which the methods of an object may be executed. In this paper, such a specification is called the *life cycle* of an object. OMT, OOD, and UML use state-transition diagrams for specifying life cycles. Such a diagram shows the state space of an

* This work was done while the author was employed at the Department of Computing Science, Eindhoven University of Technology, The Netherlands.

object and the method calls that cause a transition from one state to another. Looking at the definition of inheritance in, for example, OOD and its informal explanation, a subclass that inherits features of some other class extends the *static structure* as well as the *dynamic behavior* of its parent class. However, in the further treatment of inheritance, OOD only defines inheritance of static features. It does not specify the meaning of inheritance of dynamic behavior. It is implicitly assumed that the behavior of a subclass is an *extension* of the behavior of its superclass. OOD does not further elaborate on the precise meaning of "extension."

In this paper, we use Petri nets (See for example [11]) for specifying the dynamics of an object class. There are several reasons for using Petri nets. First of all, Petri nets provide a graphical description technique which is easy to understand and close to state-transition diagrams. Second, concurrency and synchronization are easy to model in terms of a Petri net. Third, many techniques and software tools are available for the analysis of Petri nets. Finally, Petri nets have been extended with data (color), time and hierarchy [8, 9]. The extension with data allows for the modeling of attributes and methods. The extension with time allows for the quantification of the dynamic behavior of an object. The hierarchy concept can be used to structure the dynamics of an object class.

In the Petri-net framework, we formalize what it means for an object life cycle to extend another life cycle. The results presented here are based on two earlier papers. In [3], inheritance of dynamic behavior is studied in terms of a simple process algebra. We believe that the essence of inheritance of dynamic behavior consists of *blocking* and *hiding* method calls, two notions which are well investigated in process algebra. In [1], the results from [3] are translated to Petri nets. In the current paper, we do not repeat all the obtained results. We restrict ourselves to the basic definitions concerning life-cycle inheritance and an informal explanation of some important results, being four transformation rules that allow designers to construct subclasses from some given superclass in a straightforward way. The main contribution of this paper is that it shows the validity of our approach by means of a small case study. The concepts as defined in this paper are applied to the development of a groupware editor. The results show that life-cycle inheritance stimulates the reuse of life-cycle specifications. A detailed study of the notion of inheritance of dynamic behavior, further elaborating on the results of [1, 3] and the current paper, has appeared in [2].

The remainder of this paper is organized as follows. Section 2 introduces the notions of an object life cycle and inheritance of life cycles. Section 3 presents several inheritance-preserving transformation rules on object life cycles. In Section 4, the results are applied to the development of a groupware editor. Finally, Section 5 ends with some concluding remarks.

2 Object Life Cycles and Inheritance

Place/Transition nets. In this paper, we use a specific class of Petri nets known as Place/Transition nets or simply P/T nets. Let A be some universe of action labels, which in the remainder can be thought of as method identifiers. Action labels in A are the so-called *observable* actions. To denote *silent* or *unobservable* behavior, usually corresponding to internal method invocations, a special label τ is introduced. Let A_τ denote the union of A and $\{\tau\}$.

Definition 1 (Labeled P/T net). *An A_τ-labeled Place/Transition net N is a tuple (P, T, F, ℓ), where*

i) P *is a finite set of* places*;*
ii) T *is a finite set of* transitions *such that P and T are disjoint;*
iii) $F \subseteq (P \times T) \cup (T \times P)$ *is a set of directed arcs, called the* flow relation*;*
iv) $\ell : T \to A_\tau$ *is a labeling function.*

Some place p is called an *input place* of a transition t if and only if there exists a directed arc from p to t. Place p is called an *output place* of t if and only if there exists a directed arc from t to p. We use $^\bullet t$ to denote the set of input places for a transition t. The notations $t^\bullet$, $^\bullet p$, and $p^\bullet$ have similar meanings.

The following definition introduces a property of P/T nets that is useful in the definition of object life cycles. The reflexive and transitive closure of a relation R is denoted R^*; the inverse of R is denoted R^{-1}.

Definition 2 (Connectedness). *A P/T net (P, T, F, ℓ) is* connected*, if and only if, for every two places or transitions $x, y \in P \cup T$, $(x, y) \in (F \cup F^{-1})^*$.*

Places of a P/T net may contain zero or more *tokens*. The *state* or *marking* of a net is the distribution of tokens over the places. A marking is represented by a finite multi-set, or bag, of places. The following notations are used for bags. For the explicit enumeration of a bag, a notation similar to the notation for sets is used, but using square brackets instead of curly brackets and using superscripts to denote the cardinality of the elements. To denote individual elements of a bag, the same symbol "$\in$" is used as for sets. The sum of two bags X and Y is denoted $X + Y$; the difference of X and Y is denoted $X - Y$.

Definition 3 (Marked P/T net). *A* marked *P/T net is a pair (N, s), where N is a labeled P/T net (P, T, F, ℓ) and where s is a bag over P denoting the state of the net.*

Marked P/T nets have a dynamic behavior which is defined as follows. A transition is *enabled* if and only if each of its input places contains at least one token. An enabled transition can *fire*. If a transition fires, then it *consumes* one token from each of its input places; it *produces* one token for each of its output places. The visible effect of a firing is the *label* of the transition.

Definition 4 (Firing rule). *Let (N, s) be a marked P/T net with N equal to (P, T, F, ℓ). For any* enabled *transition $t \in T$, the firing of t is denoted $(N, s)\,[\ell(t)\rangle\,(N, s - {}^\bullet t + t^\bullet)$.*

The firing rule allows us to define the notion of reachable states.

Definition 5 (Reachability). *Let (N, s) be a marked A_τ-labeled P/T net. State s' is reachable from s, denoted $(N, s)\,[*\rangle\,(N, s')$, if and only if s' equals s or if for some $n \in \mathbb{N}$ there exist $a_0, \ldots, a_n \in A_\tau$ and markings $s_1, \ldots, s_n$ such that $(N, s)\,[a_0\rangle\,(N, s_1)\,[a_1\rangle \ldots [a_n\rangle\,(N, s')$.*

Since we are interested in comparing object life cycles, which are specified by P/T nets, it is necessary to have an equivalence for P/T nets. Nets with the same external behavior, but with possibly different silent behavior must be considered equal. For this purpose, *branching bisimilarity* is a suitable equivalence [7]. In this paper, the details of the definition of branching bisimilarity are not important and, hence, omitted. The reader is referred to [1].

Object life cycles. Using P/T nets for specifying object life cycles allows us to specify a partial ordering of method calls. However, not every labeled P/T net specifies a life cycle. A life cycle is a net having exactly one *initial* or *input* place i. It also has a unique initial transition t_{cr} which corresponds to the creation of an object. A life cycle refers to a *single object*. It suffices to consider just one object because multiple objects of the same class interact via the execution of methods and not directly via the life cycle. Since we focus on one object at a time, initially, place i contains a single token.

An object life cycle must also have a unique *final* or *output* place o. An object terminates when, and only when, it reaches the marking consisting of a single token in o. In addition, if a marking has a token in o, it must be the only token in the marking. This means that upon termination of an object, all information about the object is removed. Furthermore, we assume that it is always possible to terminate. However, this does not mean that an object is *forced* to terminate.

A final requirement of a life cycle is that it is connected. Given the other requirements, it simply makes no sense to add unconnected parts to a life cycle.

The following definition formalizes the notion of an object life cycle.

Definition 6 (Object life cycle). *Let (N, s) be a marked A_τ-labeled P/T net, where N equals (P, T, F, ℓ). Let s' be any marking reachable from s. (N, s) is an object life cycle if and only if the following conditions are satisfied.*

i) Object creation: P contains a place i and T a transition t_{cr} such that $^\bullet i = \emptyset$, $i^\bullet = \{t_{cr}\}$, and $^\bullet t_{cr} = \{i\}$;
ii) Single-object requirement: $s = [i]$;
iii) Object termination: P contains a place o such that $o^\bullet = \emptyset$; furthermore, if $o \in s'$, then $s' = [o]$;
iv) Termination option: $(N, s')\,[*\rangle\,(N, [o])$;
v) Connectedness: N is connected.

Figure 1 shows the, very simple, life cycle of a class `person`. A person comes into existence when he or she is born. While alive, a person may celebrate his or her birthday. Eventually, a person dies.

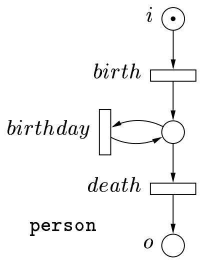

Fig. 1. A simple example of an object life cycle.

Life-cycle inheritance. At this point, it is possible to define inheritance of object life cycles. As we show in the remainder, two basic concepts play an important role in life-cycle inheritance. The first notion is the concept of *blocking* method calls. Let $(N_0, [i])$ and $(N_1, [i])$ be two object life cycles such that the set of methods of N_1 contains all methods of N_0. One possible, informal definition of inheritance is as follows.

> If it is not possible to distinguish the observable behavior of $(N_0, [i])$ and $(N_1, [i])$ when only methods of N_1 that are also present in N_0 are executed, then $(N_1, [i])$ is a subclass of $(N_0, [i])$.

In other words, $(N_1, [i])$ is a subclass of $(N_0, [i])$ if the observable behaviors of $(N_1, [i])$ and $(N_0, [i])$ are equivalent when methods of N_1 which are not present in N_0 are blocked.

The second basic concept is the concept of *hiding* method calls. Hiding a method call means that it is no longer observable. The notion of hiding inspires another definition of inheritance.

> If it is not possible to distinguish the external behavior of $(N_0, [i])$ and $(N_1, [i])$ when arbitrary methods of N_1 are executed, but when only the effects of methods that are also present in N_0 are considered, then $(N_1, [i])$ is a subclass of $(N_0, [i])$.

This means that $(N_1, [i])$ is a subclass of $(N_0, [i])$ if the observable behaviors of $(N_1, [i])$ and $(N_0, [i])$ are equivalent when hiding methods of N_1 which are not present in N_0.

The subtle difference between the two forms of inheritance is that in the second definition methods new in N_1 are executed without taking into account their effect, whereas in the first definition they are not executed at all. Below, we give an example to illustrate this difference.

To formalize inheritance of life cycles, the notions of blocking and hiding method calls must be translated to P/T nets. The technical terms for blocking and hiding actions are borrowed from process algebra, where they are called *encapsulation* and *abstraction*, respectively (see also [3]). Note that, in this paper, these two terms have a more specific meaning than usual in object-oriented design.

Definition 7 (Encapsulation and abstraction). *Assume that* (N, s) *is a marked* A_τ*-labeled P/T net with* $N = (P, T, F, \ell)$.

i) For any $H \subseteq A$*, the* encapsulation *operator* ∂_H *removes all transitions with a label in* H *from* N*. Formally,* $\partial_H(N, s) = (N', s)$ *where* $N' = (P, T', F', \ell')$ *such that* $T' = \{t \in T \mid \ell(t) \notin H\}$, $F' = F \cap ((P \times T') \cup (T' \times P))$*, and* $\ell' = \ell \cap (T' \times A_\tau)$.
ii) For any $I \subseteq A$*, the* abstraction *operator* τ_I *renames all transition labels in* I *to* τ*. That is,* $\tau_I(N, s) = (N', s)$ *where* $N' = (P, T, F, \ell')$ *such that for any* $t \in T$, $\ell(t) \in I$ *implies* $\ell'(t) = \tau$ *and* $\ell(t) \notin I$ *implies* $\ell'(t) = \ell(t)$.

Note that blocking a method is achieved by removing the corresponding transitions from the net structure.

Encapsulation and abstraction can be used to define four different inheritance relations between object life cycles. Of course, it is possible to formalize the above two informal definitions of inheritance. It is also possible to combine encapsulation and abstraction in the definition of inheritance. In this way, two more meaningful inheritance relations can be defined. For a detailed study of the four inheritance relations, the reader is referred to [1–3]. In this paper, we focus on the most general inheritance relation, called *life-cycle inheritance*. An object life cycle is a subclass of another object life cycle, its superclass, if and only if hiding some methods and blocking some other methods of the subclass results in an object life cycle equivalent to its superclass.

Definition 8 (Life-cycle inheritance). *Let* $(N_0, [i])$ *and* $(N_1, [i])$ *be two object life cycles; let* A *be the set of observable method identifiers. Life cycle* $(N_1, [i])$ *is a subclass of life cycle* $(N_0, [i])$ *under* life-cycle inheritance *if and only if there exist* disjoint $H \subseteq A$ *and* $I \subseteq A$ *such that* $\tau_I \circ \partial_H(N_1, [i])$ *is branching bisimilar to* $(N_0, [i])$.

As mentioned, in [1–3], life-cycle inheritance is investigated in detail. Among other things, it is shown that it has several desirable properties such as transitivity and reflexivity. Figure 2 gives four object life cycles of persons illustrating the definition of life-cycle inheritance.

Class `person1` describes a person that, at some point during his or her life, decides to marry. A person that can decide whether to marry or not, as described by class `person2`, should be a subclass of `person1`. It is easy to see that *blocking* method *stay_single* means that the behavior of the two persons is identical. Hence, class `person2` is a subclass of `person1`. Note that *hiding* the method *stay_single* does not yield the desired result. When hiding *stay_single* instead of blocking it, a possible behavior of a person of class `person2` is *birth* followed by *death*. A person of class `person1` can never exhibit such a behavior.

Class `person3` shows a person that divorces after marriage. Hiding the new method *divorce* easily shows that `person3` is a subclass of `person1`. In this case, *blocking* the new method in the subclass does not yield the desired result. Blocking *divorce* means that a person of class `person3` will never die, which is clearly not true for persons of class `person1`.

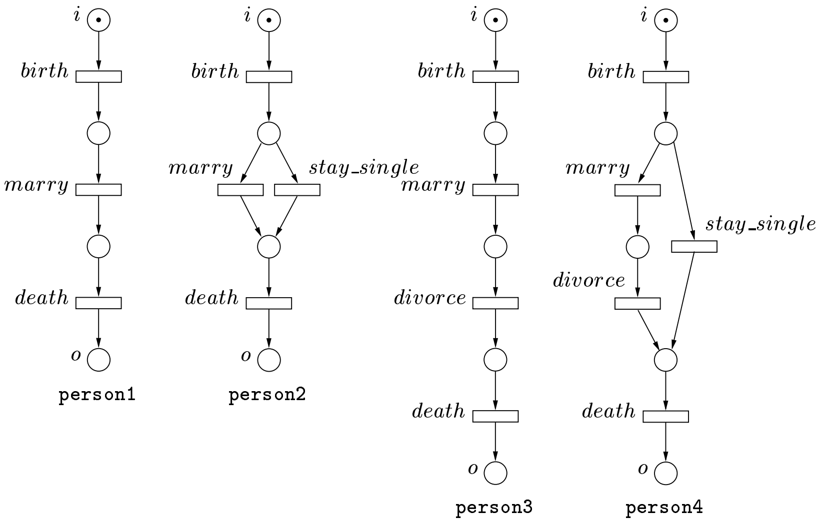

Fig. 2. An illustration of life-cycle inheritance.

Class `person4` extends class `person1` with both methods *stay_single* and *divorce*. In this example, the *combination* of encapsulation and abstraction is needed to show that `person4` is a subclass of `person1`.

3 Inheritance-Preserving Transformation Rules

It is possible to show that object life cycles can only have finitely many states. In Petri-net terminology, they are bounded [1]. This implies that checking branching bisimilarity between life cycles is decidable. However, in practical applications, it can be a complex and tedious task. Therefore, this section presents several inheritance-preserving transformation rules. These rules can be used to design subclasses of some given class, thus, supporting reuse of life-cycle specifications. Moreover, they show the essence of life-cycle inheritance. In this paper, we only give an informal explanation of the transformation rules. Formal definitions can be found in [1]. In [2], inheritance-preserving transformation rules are studied in more detail. In particular, attention is paid to developing a set of transformation rules that is effective in practical design situations.

Let N_q be a P/T net such that $(N_q, [i])$ is an object life cycle. Figure 3 shows four inheritance-preserving transformation rules transforming N_q into a subclass. For the sake of simplicity, we require that the result of each transformation is again an object life cycle.

The first transformation rule, for reasons explained in [3] called PT, can be formulated as follows. The *alphabet* of a P/T net denotes the set of all transition labels occurring in the net which are not equal to τ.

If N_p is a P/T net such that *i*) no transitions are shared between N_p and N_q and *ii*) all transitions of N_p with input places in N_q have a label which does

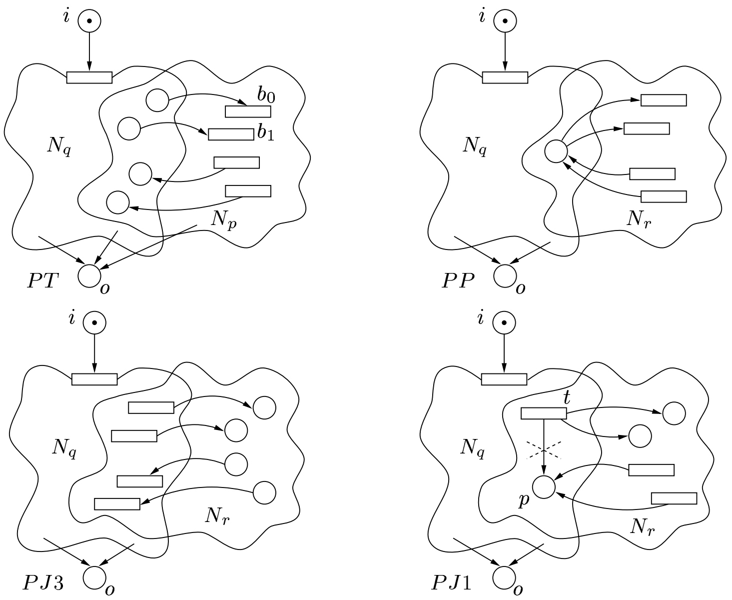

Fig. 3. Inheritance-preserving transformation rules.

not appear in the alphabet of N_q and is not equal to τ, then the union of N_p and N_q is a subclass of N_q.

The above transformation rule means that it is allowed to add a *choice*, or a new branch of behavior, to an existing object life cycle. In the example of Figure 3, it is crucial that methods b_0 and b_1 do not occur in the alphabet of N_q. It is not difficult to see that blocking b_0 and b_1 leads to a net whose behavior is equivalent to the behavior of N_q. Methods b_0 and b_1 act as so-called *guards*, separating the subclass extension from the original life cycle. In the example of Figure 2, the extension of class `person1` with method *stay_single* resulting in class `person2` is captured by transformation rule PT.

The second rule shown in Figure 3, called PP, is a special case of PT showing that PT captures a simple form of *recursion*. Rule PP states that it is possible to extend an existing class with an iteration. Under certain assumptions about the net N_r, it preserves a more restricted form of inheritance than PT does (see [1, 2] for details).

The third transformation rule, called $PJ3$ shows that it is possible to add *parallel behavior* to an object life cycle.

If N_r is a P/T net such that *i*) no places are shared between N_q and N_r, *ii*) all transitions of N_r have a label which does not appear in the alphabet of N_q and *iii*) transitions in N_q with input places in N_r obey the free-choice property, then the union of N_r and N_q is a subclass of N_q.

The second requirement means that only new methods are used in the extension of the original life cycle. This means that hiding the new methods yields a behavior equivalent to that of the original life cycle. The third requirement is needed for the correctness of the transformation rule. In most practical examples, such as the examples in the next section, it is satisfied. An introduction to the free-choice property and the class of free-choice Petri nets can be found in [5].

The fourth and final transformation rule, $PJ1$, shows that it is possible to *insert* behavior in between two parts of the original life cycle.

> If N_r is a net such that *i)* N_q and N_r share exactly one place p and one transition t with (t, p) in the flow relation of N_q, *ii)* all transitions of N_r other than t have a label which does not occur in the alphabet of N_q, and *iii)* N_r satisfies certain requirements with respect to liveness, boundedness, and the free-choice property, then the union of N_q and N_r *without* the arc (t, p) is a subclass of N_q.

This transformation rule says that it is allowed to replace an arc in an object life-cycle by an entire P/T net. Again, the requirement that all new methods must have a fresh identifier means that hiding these methods yields a behavior equivalent to the behavior of the original life cycle. The third requirement above is necessary to guarantee that the extension behaves properly. That is, if transition t fires and thus activates the extension N_r, it must be possible that eventually a token arrives in p without leaving any tokens in the other places of N_r. This means that the behavior of the original net N_q is not affected by the addition of N_r. By applying rule $PJ1$, it is possible to show that class `person3` in Figure 2 is a subclass of class `person1` in the same figure.

The four transformation rules presented in this section are not the only possible ones. However, they are characteristic for life-cycle inheritance. It is interesting to note that they capture important design constructs such as choices (PT), parallelism ($PJ3$), sequential composition ($PJ1$), and iteration (PP). This observation in particular has convinced us that the concepts presented in this paper touch upon the fundamentals of inheritance of dynamic behavior.

4 The Groupware Editor

Introduction. In this section, the concepts developed in the previous sections are applied to the development of a groupware editor. We do not give full specifications of the classes involved, but focus on the object life cycles. This means that we do not specify data types, class attributes, or method implementations. If necessary, an informal explanation is given.

The requirements for the groupware editor are as follows. The editor is meant to edit some kind of diagrams. Multiple users, possibly situated at different workstations, may be editing a single diagram in a joint editing session. Users may choose to either view or edit a diagram. They may join or leave a session at will. It must be clear to all users who is currently editing some given diagram. The diagrams under consideration may be complex, possibly consisting of multiple components. Components may introduce hierarchy in a diagram. It is possible to open or close a component revealing or hiding its details. Not just any user may view or edit any component in a diagram. Users must have permission to do so. Permissions are not fixed; to get permission, a user can simply select a component and then try the desired command. If there are no conflicting permissions, the command succeeds; otherwise, it fails and the user is notified. Users may explicitly surrender permissions. Most permissions are automatically reset if the user leaves the editing session; a few permissions may persist between sessions.

In the next paragraphs, life-cycle inheritance is used to design a groupware editor satisfying the abovementioned requirements. The development is split into three steps. First, a groupware *viewer* is designed. The design is fairly simple and the resulting system allows multiple users to view existing diagrams. Second, the viewer is transformed into a multi-user editor by adding editing functions to classes in the viewer design. Third, the multi-user editor is specialized to a groupware editor by adding permissions. The example shows how life-cycle inheritance can be used to structure an object-oriented design process. It also shows how object life cycles can be reused in a design.

The multi-user viewer. In the design of the viewer, four classes can be distinguished: `user`, `diagram`, `user_session`, and `edit_session`. The first two classes have straightforward interpretations. The third and fourth class are intended to structure viewing sessions. An `edit_session` object keeps track of all users viewing a particular diagram; an object of class `user_session` maintains all data involved in a particular session of a particular user. Figure 4 shows the life cycles for objects of the above four classes.

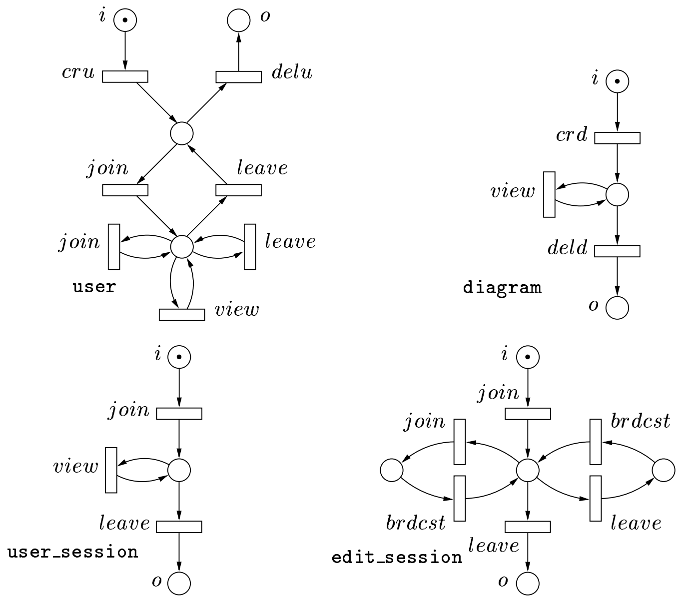

Fig. 4. A multi-user viewer.

Users can be created (*cru*) or deleted (*delu*). This means that they are added to or removed from the list of users of the viewer. Once a user exists, (s)he may join and leave editing sessions. If the user participates in at least one edit session, (s)he may issue *view* commands in order to view diagrams. It is not difficult to see that `user` satisfies the requirements of an object life cycle (Definition 6).

Class `diagram` has a straightforward life cycle. Diagrams can be created (*crd*), viewed (*view*), and deleted (*deld*). A user interacts with a diagram through a `user_session` object. Upon joining an editing session, a new object of class `user_session` is created. The only possible command a user can execute is the *view* command. If the user leaves the editing session, the `user_session` object is terminated.

Objects of class `edit_session` keep track of all users involved in a single session. The first user that "joins" an editing session for some diagram, actually creates a new `edit_session` object. If other users join a running session, this does not lead to the creation of a new object. The information is simply stored in the existing object. To fulfill the requirement that all users must know who is participating in the session, we assume that the implementation of *join* is such that the new user gets a list of users already present. Furthermore, the information about a new user is broadcast (*brdcst*) to all other users participating in the session. When a user leaves, this information is broadcast to all remaining users. The last user leaving the session terminates the `edit_session` object.

In a complete implementation of a system, it is clear from the code which methods interact with each other. Since we do not give method implementations, we make a few assumptions. First, `user` methods invoke methods with the same label in `user_session`, which, in turn, invoke methods of the same name in `edit_session` and `diagram`. Second, methods which do not have counterparts in one of the other classes are assumed to interact with (objects in) the environment. Examples are *cru*, *delu*, *crd*, *deld*, and *brdcst*.

The multi-user editor. This paragraph describes a specialization of the multi-user viewer, namely a multi-user editor. Permissions are not yet incorporated. They are added in the next paragraph. As a result, in this version of the editor, it is possible that a component is deleted by one user, while it is being changed or viewed by another one.

Editing facilities have been added to three classes, namely classes `user`, `diagram`, and `user_session`. Class `edit_session` does not need to change. The new classes are shown in Figure 5. In this paragraph, we argue that the three new classes are subclasses of the corresponding classes of the viewer. Some places in the life cycles of Figure 5 are labeled for the purpose of future reference.

Let us start with class `diagram_e`. It is now possible to modify diagrams by means of method *mod*. It follows from transformation rule PP depicted in Figure 3 that `diagram_e` is a subclass of class `diagram`. It is clear that blocking method *mod* in `diagram_e` leads to a net equivalent with `diagram`. So in this simple case, it also follows directly from the definition of life-cycle inheritance that `diagram_e` is a subclass of `diagram`.

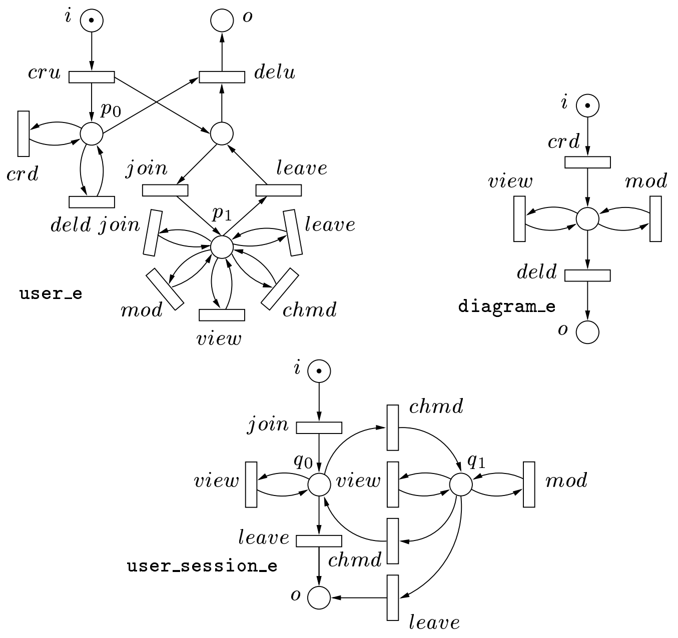

Fig. 5. A multi-user editor.

Class `user` has been extended to class `user_e`. Since we are developing an editor, users of the system are now responsible for creating and deleting diagrams. Creation and deletion of diagrams can be done independently of editing (other) diagrams. This means that we added parallel behavior to the life-cycle of `user` objects. It follows from rule $PJ3$ that the addition of place p_0 and methods *crd* and *deld* yields a subclass of `user`. A second addition is that users can now modify diagrams. For this purpose, method *mod* has been added. Users have to choose whether they want to modify a diagram or whether they are satisfied with just the option to view it. Method *chmd* (change mode) can be used to toggle between viewing and editing mode. The implementation of *mod* and *chmd* can be such that only a subset of the users is allowed to enter editing mode. It simply follows from applying transformation rule PP that the addition of methods *mod* and *chmd* to class `user` leads to a subclass. Therefore, the subsequent application of rule $PJ3$ and rule PP yields that `user_e` is a subclass of `user`.

For each editing session a user joins, a `user_session_e` object is created. This object keeps track of the mode in which the user is for this particular diagram. Initially, the user is in viewing mode. By invoking method *chmd* the user can change to editing mode. This means that we have extended the life cycle of `user_session` objects with a choice. Transformation rule PT can be applied to show that `user_session_e` is a subclass of `user_session`. Method *chmd* acts as the guard.

Summarizing, in this paragraph, we have applied three of the four transformation rules given in Section 3 to extend the viewer of the previous paragraph to an editor. What is important is that we have *reused* the specifications for the object life cycles of the viewer in the design process.

The groupware editor. In this paragraph, we extend the multi-user editor of the previous paragraph with permissions. Permissions are needed to prevent all kind of anomalies. For example, they guarantee that it is impossible that one user deletes a component when another user is viewing it. Four new classes are introduced, namely `user_p`, `diagram_p`, `user_session_p`, and `edit_session_p`. Figure 6 shows two of these four classes.

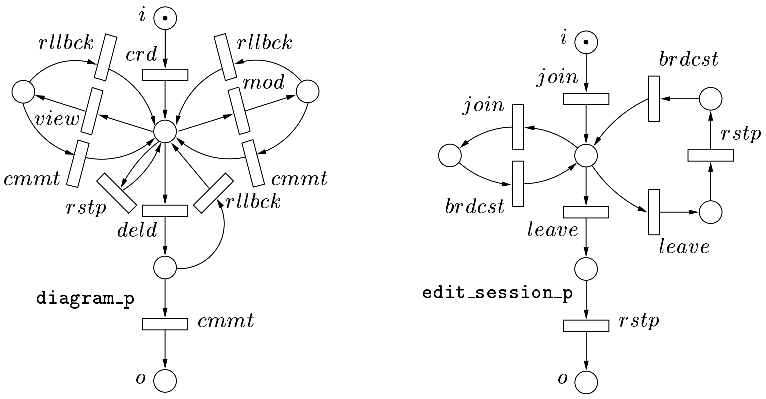

Fig. 6. A groupware editor.

The current set of permissions for editing a diagram is maintained in the corresponding `diagram_p` object. This is the only feasible option since some permissions may persist between editing sessions. As stated in the requirements, a user can get permission for some editing command by simply selecting a component and executing the command. Therefore, after every editing action *view*, *mod*, or *deld*, `diagram_p` executes either a rollback (*rllbck*) or a commit (*cmmt*), depending on whether or not the user has permission for the editing action. Another change when compared to class `diagram_e` is that the method *rstp* has been added which can be used to reset permissions. The three other classes must invoke *rstp* whenever appropriate.

Classes `user_p` and `user_session_p` are simple extensions of `user_e` and `user_session_e` and, therefore, not shown in Figure 6. Since users may surrender permissions at any time, class `user_p` is obtained from `user_e` of Figure 5 by adding a loop consisting of a single transition labeled *rstp* to place p_1. Since users interact with diagrams through user sessions, the life cycle of `user_session_p` is constructed from the life cycle of class `user_session_e` by adding the same loop as above to places q_0 and q_1.

Class `edit_session_p` is constructed from class `edit_session` of Figure 4. As mentioned, permissions are reset when a user leaves a session. This means that `edit_session` is extended with calls of method *rstp* after each invocation of the *leave* method. The resulting class `edit_session_p` is shown in Figure 6.

It remains to be shown that the new classes are subclasses of the corresponding classes in the earlier designs. The addition of method *rstp* to classes `user_e` and `user_session_e` is captured by transformation rule PP. Hence, `user_p` and `user_session_p` are subclasses of classes `user_e` and `user_session_e`, respectively.

The addition of *rstp* to `diagram_e` is also captured by rule PP. Applying rule $PJ1$ three times shows that the addition of *cmmt* to `diagram_e` preserves life-cycle inheritance. To show that the addition of *rllbck* preserves inheritance, consider the intermediate result obtained after the previous additions. The desired result now follows easily from transformation rule PT. Hence, `diagram_p` is a subclass of `diagram_e` and, therefore, also of `diagram`.

To show that `edit_session_p` is a subclass of `edit_session`, we have to apply rule $PJ1$ twice simultaneously.

5 Concluding Remarks

In this paper, we have presented a definition for inheritance of dynamic behavior in the framework of Petri nets. In our opinion, the notions of *blocking* and *hiding* method calls are fundamental to inheritance of dynamic behavior. We have informally presented four inheritance-preserving transformation rules that allow for the straightforward construction of subclasses from a given superclass. To validate our approach, we have applied the transformation rules to the development of a groupware editor. However, we only considered life cycles. We did not give full class definitions nor did we give implementations of all the methods. A future challenge is to incorporate the results either in a full fledged object-oriented formalism based on Petri nets, such as OPN [10], or to incorporate them into a framework as OMT, OOD, or UML. In the latter case, one could choose to translate the notion of life-cycle inheritance to state-transition diagrams or one could choose to replace state-transition diagrams by Petri nets. An advantage of incorporating life-cycle inheritance into a Petri-net-based formalism as OPN is that one obtains an integrated framework with a sound theoretical basis. A disadvantage is that object-oriented languages based on Petri nets are not yet in common use. An advantage of incorporating the results in a framework as OMT, OOD, or UML is that it will be easier to get acceptance of the notion of life-cycle inheritance in practice, particularly when it is translated to state-transition diagrams.

References

1. W.M.P. van der Aalst and T. Basten. Life-Cycle Inheritance: A Petri-Net-Based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997, 18th. International Conference, ICATPN'97, Proceedings*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81, Toulouse, France, June 1997. Springer, Berlin, Germany, 1997.

2. T. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra.* PhD thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, December 1998.
3. T. Basten and W.M.P. van der Aalst. A Process-Algebraic Approach to Life-Cycle Inheritance: Inheritance = Encapsulation + Abstraction. Computing Science Report 96/05, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, March 1996.
4. G. Booch. *Object-Oriented Analysis and Design: With Applications.* Benjamin/Cummings, Redwood City, CA, USA, 1994.
5. J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, UK, 1995.
6. M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, Reading, Massachusetts, USA, 1997.
7. R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics (extended abstract). In G.X. Ritter, editor, *Information Processing 89: Proceedings of the IFIP 11th. World Computer Congress*, pages 613–618, San Fransisco, California, USA, August/September 1989. Elsevier Science Publishers B.V., North-Holland, 1989.
8. K.M. van Hee. *Information Systems Engineering: A Formal Approach.* Cambridge University Press, Cambridge, UK, 1994.
9. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use,* volume 1*, Basic Concepts.* EATCS monographs on Theoretical Computer Science. Springer, Berlin, Germany, 1992.
10. C. Lakos. From Coloured Petri Nets to Object Petri Nets. In G. De Michelis and M. Diaz, editors, *Application and Theory of Petri Nets 1995, 16th. International Conference, Proceedings*, volume 935 of *Lecture Notes in Computer Science*, pages 278–297, Torino, Italy, June 1995. Springer, Berlin, Germany, 1995.
11. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS monographs on Theoretical Computer Science.* Springer, Berlin, Germany, 1985.
12. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.